DW_lp_fifo_1c_df

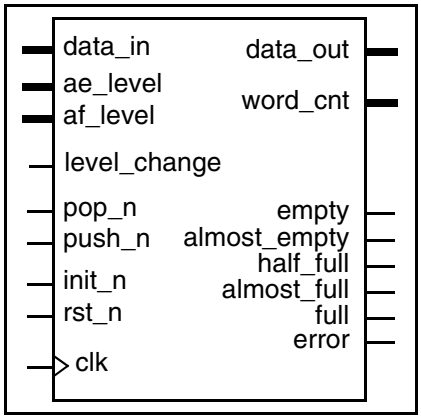
Low Power Single Independent Clock FIFO

Version, STAR, and myDesignWare Subscriptions: IP Directory

Features and Benefits

Revision History

- RAM bypassing using pre-fetch cache structure
- Single clock cycle execution on all operations
- Low power features and design techniques applied
- Fully registered synchronous address and flag output ports
- Dynamically programmable almost full and almost empty flags
- Interfaces to common hard macro or compiled ASIC dual-port synchronous RAMs
- Parameterized RAM size
- Push error (overflow) and pop error (underflow) flags



Description

DW_lp_fifo_1c_df is a single-clock FIFO that contains a dual-port synchronous RAM along with the DesignWare component DW_lp_fifoctl_1c_df. Word caching (or pre-fetching) is performed in the pop interface to minimize latencies via a RAM by-pass feature, allow for bursting of contiguous words, and provide registered data to the external logic. The caching depth is configurable from 1 to 3, depending on the synchronous RAM configuration. This component contains complete flexibility in interfacing with all configurations of synchronous RAM.

Supported synchronous RAM architectures:

- Non re-timed write port and asynchronous read port
- Re-timed write port and asynchronous read port
- Non re-timed write port and synchronous read port with buffered read address and non-buffered read data
- Non re-timed write port and synchronous read port with non-buffered read address and buffered read data
- Non re-timed write port and synchronous read port with buffered read address and buffered read data
- Re-timed write port and synchronous read port with buffered read address and non-buffered read data
- Re-timed write port and synchronous read port with non-buffered read address and buffered read data

- Re-timed write port and synchronous read port with buffered read address and buffered read data

The FIFO provides parameterized data width, FIFO depth and data pre-fetching cache depth that are all configurable upon module instantiation.

As an extra level of flexibility, the DW_lp_fifo_1c_df is configurable to allow write path re-timing (push interface) and/or no pre-fetching cache to model predecessor DesignWare library components.

Table 1-1 Pin Description

Pin Name	Width	Direction	Function
clk	1 bit	Input	Input clock
rst_n	1 bit	Input	Asynchronous reset (active low)
init_n	1 bit	Input	Synchronous reset (active low)
ae_level	ceil($\log_2$[*depth*] + 1)	Input	Almost empty level for the `almost_empty` output (the number of words in the FIFO at or below which the `almost_empty` flag is active)
af_level	ceil($\log_2$[*depth*] + 1)	Input	Almost full level for the `almost_full` output (the number of empty memory locations in the FIFO at which the `almost_full` flag is active)
level_change	1 bit	Input	Enable update of almost_empty and/or almost_full state when `ae_level` and/or `af_level` change
push_n	1 bit	Input	Push request (active low)
data_in	width	Input	Input data
pop_n	1	Input	Pop request (active low)
data_out	*width*	Output	Output data
word_cnt	ceil(log2(*depth* + 1))	Output	FIFO word count
empty	1 bit	Output	FIFO empty flag
almost_empty	1 bit	Output	Almost empty flag (determined by `ae_level` input)
half_full	1 bit	Output	Half full flag
almost_full	1 bit	Output	Almost full flag (determined by `af_level` input)
full	1 bit	Output	Source domain RAM full flag
error	1 bit	Output	Error flag (overrun or underrun)

Table 1-2 Parameter Description

Parameter	Values	Description
width	1 to 1024 Default: 8	Vector width of data bus to/from RAM
depth	4 to 1024 Default: 8	Depth of FIFO
mem_mode	0 to 7 Default: 3	RAM configuration Identifies where and how many re-timing stages needed in RAM which determines pre-fetch cache buffering depth: ■ 0: No retiming stages ■ 1: RAM data out re-timing ■ 2: RAM read address re-timing ■ 3: RAM data out and read address re-timing ■ 4: RAM write interface re-timing ■ 5: RAM write interface and RAM data out re-timing ■ 6: RAM write interface and read address re-timing ■ 7: RAM write interface, read address, and read address re-timing
arch_type	0 to 4 Default: 1	Datapath architecture configuration ■ 0: No input re-timing, no pre-fetch cache (*mem_mode* must be set to 0 for this setting of *arch_type*) ■ 1: No input re-timing, pipeline pre-fetch cache ■ 2: Input re-timing, pipeline pre-fetch cache ■ 3: No input re-timing, register file pre-fetch cache ■ 4: Input re-timing, register file pre-fetch cache For details about *arch_type*, see “Detailed Description of Parameter arch_type” on page 5
af_from_top	0 or 1 Default: 1	Almost full level input (`af_level`) usage ■ 0: The `af_level` input value represents the minimum number of valid FIFO entries at which the `almost_full` output starts being asserted ■ 1: The `af_level` input value represents the maximum number of unfilled FIFO entries at which the `almost_full` output starts being asserted For details about *af_from_top*, see “Detailed Description of Parameter af_from_top” on page 5

Table 1-2 Parameter Description (Continued)

Parameter	Values	Description
ram_re_ext	0 or 1 Default: 0	Determines the characteristic of the internal `ram_re_n` signal to RAM ■ 0: Single-cycle pulse of internal `ram_re_n` at the read event to RAM ■ 1: Extend assertion of internal `ram_re_n` while read event active in RAM
err_mode	0 or 1 Default: 0	Error reporting ■ 0: Sticky error flag ■ 1: Dynamic error flag
rst_mode	0 to 3 Default: 0	System reset mode Defines the behavior of `rst_n` and clearing of RAM: ■ 0: `rst_n` is asynchronous, RAM cleared by `rst_n` or `init_n` ■ 1: `rst_n` is asynchronous, RAM not cleared by `rst_n` or `init_n` ■ 2: `rst_n` is synchronous, RAM cleared by `rst_n` or `init_n` ■ 3: `rst_n` is synchronous, RAM not cleared by `rst_n` or `init_n`

Table 1-3 Synthesis Implementations

Implementation Name	Function	License Feature Required
rtl	Synthesis model	■ DesignWare (P-2019.03 and later) ■ DesignWare-LP[a] (before P-2019.03)

a. For versions before P-2019.03, you must enable minPower as follows:

```
set_synthetic_library {dw_foundation.sldb dw_minpower.sldb}
```

Table 1-4 Simulation Models

Model	Function
DW06.DW_LP_FIFO_1C_DF_CFG_SIM	Design unit name for VHDL simulation
dw/dw06/src/DW_lp_fifo_1c_df_sim.vhd	VHDL simulation model source code
dw/sim_ver/DW_lp_fifo_1c_df.v	Verilog simulation model source code

Detailed Description of Parameter *arch_type*

The *arch_type* parameter is available for selection of which structures will exist, if any, in the data path before and after the RAM. If *arch_type* is 0, the DW_lp_fifoctl_1c_df will route the data path and push and pop controls directly to and from the RAM. That is, no storage elements will exist in the data path of the DW_lp_fifoctl_1c_df.

When *arch_type* is non-zero, the DW_lp_fifoctl_1c_df will contain a pre-fetch cache used as a RAM bypass to minimize latencies. The depth of the pre-fetch cache is dependent on the synchronous RAM configuration. For more, see "Cache Size when arch_type is 1 through 4" on page 11.

When *arch_type* is 2 or 4, a 1-stage set of input re-timing registers is placed on `data_in` and `push_n`. This input re-timing stage enables flexibility for designs with a late-arriving write interface and provides the designer a built-in structure to meet timing constraints.

There are two pre-fetch architectures available to allow the designer to optimize for power consumption considerations; *arch_type* settings 1 and 2 versus 3 and 4. If reduction of power consumption is a system guideline, selecting the optimal pre-fetch cache architecture is dependent on the push and pop activity characteristics to the FIFO. More detail is provided in section "Pre-fetch Cache Architectures" on page 7.

Detailed Description of Parameter *af_from_top*

The *af_from_top* parameter provides the option for how the `af_level` input is interpreted and used in determining how the state of the `almost_full` output is derived. When *af_from_top* is 1 (default) the DW_lp_fifoctl_1c_df interprets the `af_level` value as the maximum number of empty locations in the FIFO from the top (or full condition) in which the `almost_full` flag is set to 1. The `af_level` value in this case is subtracted from `depth` and compared to `word_cnt`.

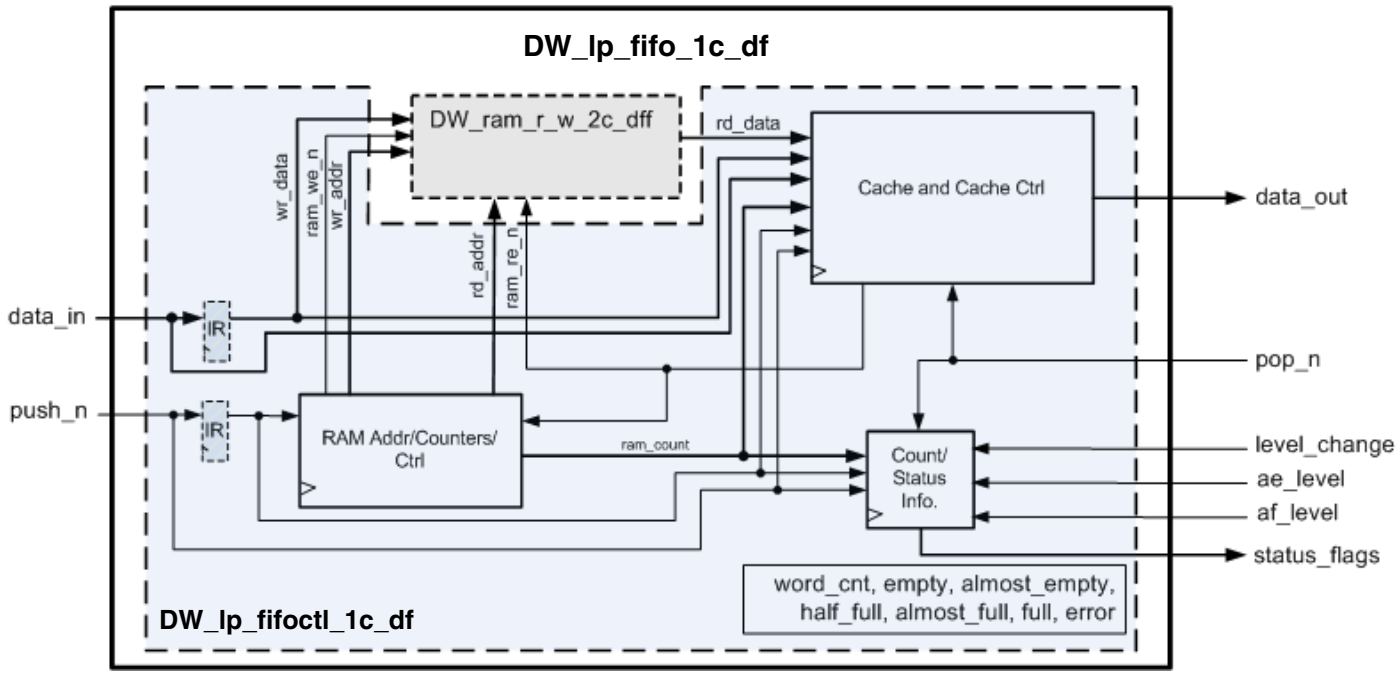
If the desire is not to have a subtraction operator built in to the DW_lp_fifoctl_1c_df with regard to determining `almost_full` and, thus, gaining a area savings, setting *af_from_top* to 0 would be the choice to make. For this setting of *af_from_top*, `af_level` is interpreted as the threshold of the word count at which `almost_full` is set to 1. Any word count equal to or greater than `af_level` would result in `almost_full` being a 1; else, it would be 0.

Examples of both settings of *af_from_top* are shown in the Timing Diagrams sections, for *af_from_top* = 0, refer to "Push to Full, Pop to Empty, and Pop Error" on page 19; for *af_from_top* = 1, refer to "Push to Full, Push and Pop While Full, Push Error" on page 21.

Block Diagram

Figure 1-1 show the block diagram of the DW_lp_fifo_1c_df:

Figure 1-1 DW_lp_fifo_1c_df Basic Block Diagram



Low Power Features

Emphasis is made on minimizing power consumption. Application of component features, RTL implementation techniques, and providing alternative functional configurations are incorporated in this component. The following are items that describe the features implemented in the underlying DW_lp_fifoctl_1c_df component that minimize power.

RAM Read Enable

This component provides a read enable to RAM to allow the RAM to disable read switching activity when no read access is occurring. In applications where popping is not continuous, shutting down the read port of the RAM during idle popping cycles assists in minimizing power. Control in the behavior of the internal RAM read enable is provided via the parameter *ram_re_ext* that may assist in minimizing power depending on the system activity.

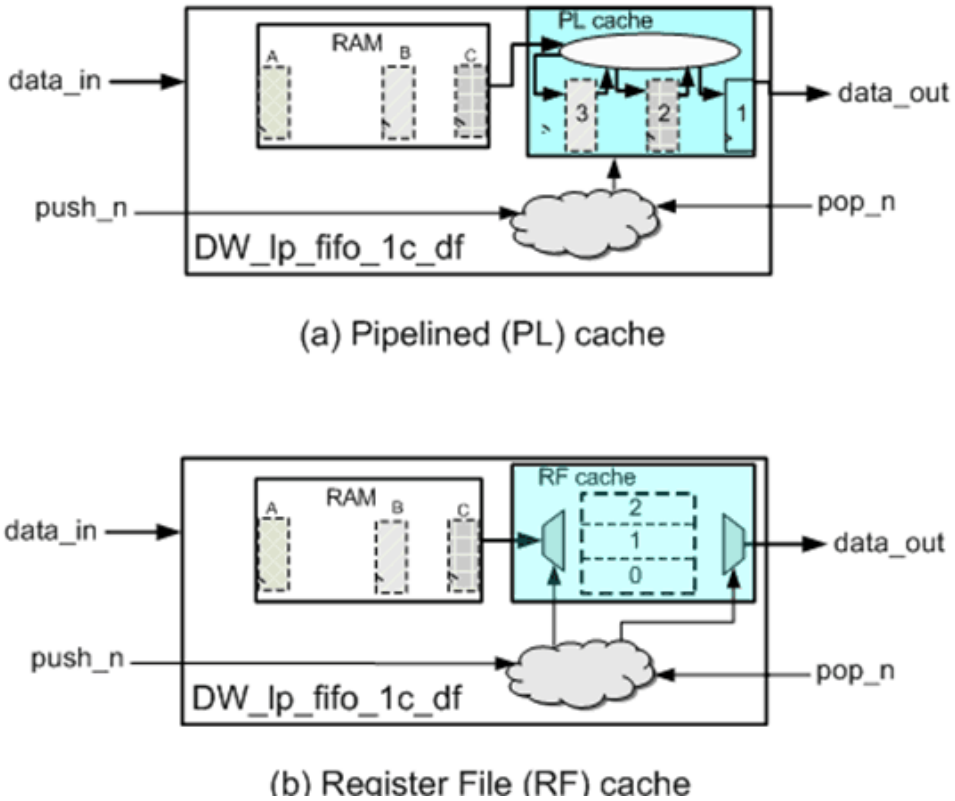
Coding Style Enabling Clock Gating

Code implementation techniques are applied throughout that enables the power compiling tool to perform clock gate insertion. This approach provides a significant power savings, especially as data widths get larger.

Pre-fetch Cache Architectures

The pre-fetch cache architectures as originally conceived are based on the DW_lp_fifoctl_2c_df which exists in the underlying component DW_lp_fifoctl_1c_df. There are two parameter-selectable pre-fetch cache architectures that allow for power optimization: pipelined (PL) and register file (RF) types. Figure 1-2 shows a block diagram of each cache style.

Figure 1-2 Block Diagram of Two Styles of Cache (shaded)



The PL caching style is effectively a shift register of 1, 2, or 3 stages. Active switching through each stage occurs during shifting initiated by pop requests with pending valid data in either the RAM or cache stages behind the head location. In cases with a wide data bus and cache configurations of 2 or 3 deep, this could represent the majority of the register switching power consumption within the component.

For cache depths of 2 or 3, an alternative pre-fetch cache is provided in the form of an RF structure. For the RF cache structure, the shifting between pipelined cache entries is eliminated and replaced with write and read pointer manipulation to access cache elements; a mini-FIFO of sorts.

The two caching architectures are provided to give the designer flexibility in selecting the caching architecture that will yield the lowest total power consumption. Knowing which pre-fetch cache architecture to choose is highly dependent on factors such as technology, clock rate, data width, pre-fetch cache depth, and data flow characteristics through the associated FIFO.

With all variables being equal, the advantage that either cache architecture provides in terms of optimal power dissipation is based particularly on the type of data flow through the DW_lp_fifoctl_1c_df.

Generally, there's no rule of thumb in selecting which cache architecture will render the least power dissipation. This may require the design process to include some experimentation in using both methods to characterize behavior based on the system parameters.

Keep in mind, criticality in choosing which pre-fetch cache architecture is only meaningful in a system when the parameter *mem_mode* is not 0. That is, when the cache depth is 2 or 3, the selection of the cache architecture becomes relevant.

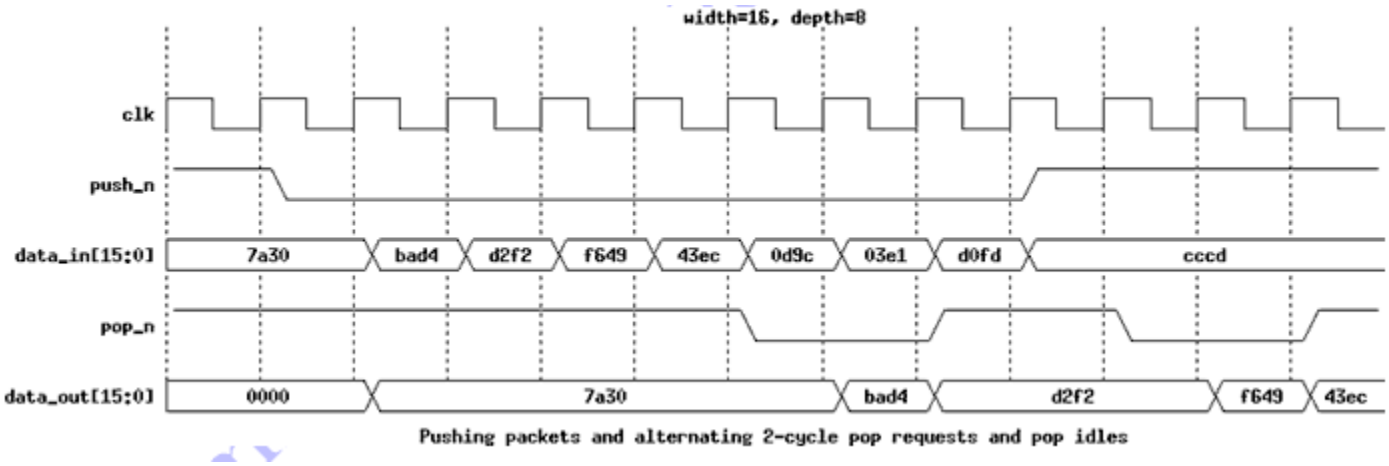
The key factor that determines which cache architecture will provide the least power consumption is the behavior of the pop requests (`pop_n`). If pop requests are issued in short bursts of 3 or less, the RF cache will most likely yield the least power consumption. If however, pop requests occur in longer contiguous bursts, Pipeline cache architecture should yield the low power consumption.

Note: A characteristic that differs with the RF cache (versus the PL cache) is its non-registered `data_out`. The `data_out` goes through a multiplexer controlled by the decoding of the read cache pointer that is two bits at most. Nonetheless, an investigation and characterization of the two cache architectures is a worthwhile exercise in determining feasibility of function and power savings.

The following two cases show two extremes in data flow behavior in which each cache architecture is selected, generally, in terms of providing optimal power consumption.

CASE 1: Push packets and pop alternately with two cycles active then two cycles inactive

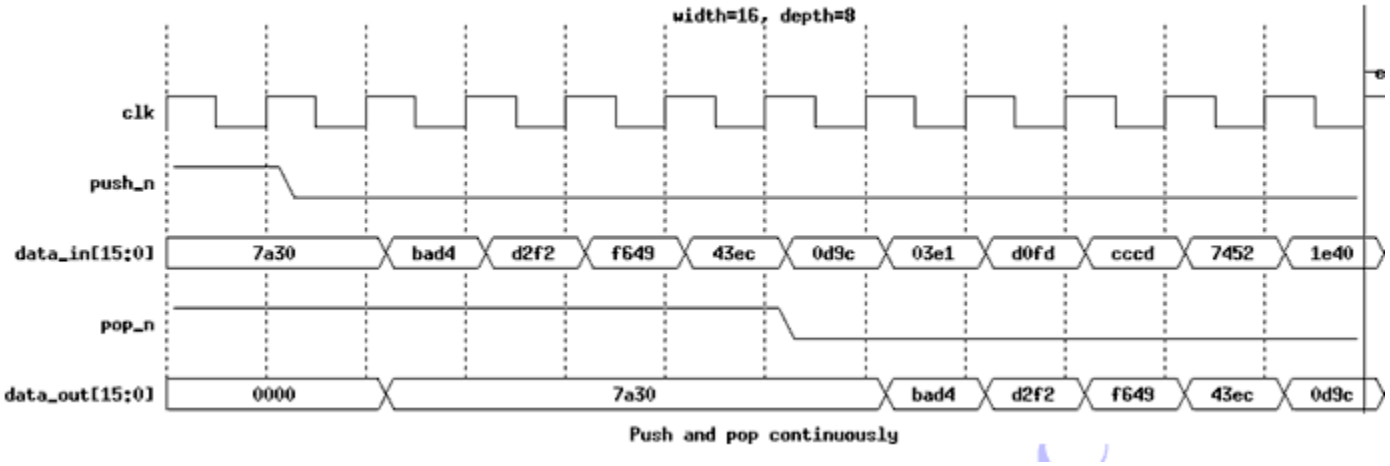
Figure 1-3 Data flow of Popping Small Bursts



The data flow behavior for Figure 1-3 shows a packet of length *depth* = 8 with data pushed contiguously before pushing halts. The pop activity begins as the FIFO is approximately half full and follows a progression of 2 cycles active and 2 cycles idle. This particular data flow, assuming cache depth is 3, is the best-case behavior that favors the selection of the Register File (RF) cache architecture (*arch_type* of 3 or 4). Similarly, the pop request was active every other cycle for cache depth of 2 (*arch_type* of 1 or 2) would be best-case data flow behavior geared to selecting the RF cache style.

CASE 2: Popping in long contiguous bursts

Figure 1-4 Data Flow of Popping Continuously



When long bursts of contiguous pop requests are issued, this type of data flow favors using the PL cache architecture. The power benefits in this data flow are because the front of the cache is continuously being written to and read from. From the PL cache perspective only stage of the cache is being used at one time and effectively the other stages of the cache are unused during this time. So, the dynamic power of shifting through many stages of the cache does not occur. In the RF cache however, data is being written to cache just like in the PL cache case, but the write and read addressing logic is always active. This activity of the write and read addressing logic is extra power consumption that the RF cache architecture has but the PL cache does not. Thus, the PL cache architecture is optimal for this type of data flow.

Note that in CASE 1, the write and read address logic is always active as well. But the data flow through the cache caused by the burst pop requests is such that the cache is full, almost full, or becoming full. Thus, all the cache locations are shifting in or out data on every cycle. Therefore, power consumption of the cache is predominantly the shifting of data and not the write and read address logic. Thus, selecting the RF cache, in general, will provide best power consumption results. The differences in favor of the RF cache over the PL cache in this data flow behavior increase as data widths increase.

Memory and FIFO Size Considerations

Depending on the constraints of the system design, the RAM size and configuration may be fixed or the FIFO depth may be the fixed. In cases where the FIFO depth is fixed and the RAM size is derived, refer to Table 1-5 and Table 1-6 in determining the proper RAM size to use when the *depth* parameter is the pre-determined.

In the cases where RAM size is fixed, for example, to a 2^n size, then the parameter *depth* must be determined based on the RAM configuration. The following tables (based on the *arch_type* setting) indicates the factors involved in calculated *depth* if RAM size is fixed.

If *arch_type* is 0, then *depth* = RAM size.

If *arch_type* is 1 or 3 (no input re-timing, pre-fetch cache exists), then:

Table 1-5 Calculation of *depth* for *arch_type* of 1 or 3

depth* value based on RAM Size and *mem_mode
depth = RAM size +1 when *mem_mode* = 0
depth = RAM Size + 2 when *mem_mode* = 1, 2, 4 or 6
depth = RAM Size + 3 when *mem_mode* = 3, 5 or 7

If *arch_type* is 2 or 4 (input re-timing and pre-fetch cache exist), then:

Table 1-6 Calculation of '*depth*' for *arch_type* of 2 or 4

***depth* value based on RAM size and mem_mode**
depth = RAM Size + 2 when *mem_mode* = 0
depth = RAM Size + 3 when *mem_mode* = 1, 2, 4 or 6
depth = RAM Size + 4 when *mem_mode* = 3, 5 or 7

Write Operations

Caching (When *arch_type* Is Not 0)

For non-zero parameter values of *arch_type*, the pop interface contains output buffering (pre-fetching cache) with the number of pipeline stages determined by the *mem_mode* parameter.

When the FIFO is empty, the first word that is pushed goes directly to the pre-fetching cache (bypassing RAM) and is available for reading (popping) in the next clock cycle. All subsequent pushes go directly into the cache until it becomes full. Once the cache is full and another push occurs without a simultaneous pop request, the first RAM location is written. All subsequent pushes will go into the RAM regardless of the cache state until the RAM becomes completely empty. At that time, pushes can begin to fill the cache first prior to any writes into RAM as before. Also, note that a simultaneous push and pop with the cache full and the RAM empty causes push data to bypass the RAM and get loaded into the cache.

The pre-fetching cache can be omitted in certain configurations as needed by the system designer with the parameter value of *arch_type* is 0. However, in omitting the pre-fetching cache, there is no guarantee of a registered output from the FIFO.

When the pre-fetching cache is configured to exist, there will always be one buffering stage in the cache which is seen at the pop interface. Table 1-7 is a list identifying the number of pre-fetching stages of the cache used based on the value of the *mem_mode* parameter.

Table 1-7 Cache Size when *arch_type* is 1 through 4

***mem_mode* values**	**Number of caching stages**
0	1
1, 2, 4 or 6	2
3, 5 or 7	3

Writing to FIFO

A write to the FIFO is executed when the `push_n` input is asserted and either

- the `full` flag is inactive,

Or:

- the `full` flag is active and
- the `pop_n` is asserted.

Thus, a push can occur even if the FIFO is full as long as the pop is executed in the same cycle.

Asserting `push_n` when `full` is not asserted (and `pop_n` is a don't care) causes the following to occur when *arch_type* is 0:

- the DW_lp_fifoctl_1c_df `ram_we_n` is asserted immediately, preparing for a write to the RAM on the next rising `clk`, and
- on the next rising edge of `clk`, `wr_addr` is incremented.

For configurations where *arch_type* is 1 or 3, asserting `push_n` when `full` and `pop_n` are not asserted causes the following to occur:

If the pre-fetch cache is not full and the RAM is empty,

- `data_in` is written directly into the cache on the next rising edge of `clk`, and
- the write address to RAM is not advanced.

If the pre-fetch cache full and the RAM is not empty OR the RAM is not empty,

- the DW_lp_fifoctl_1c_df `ram_we_n` is asserted immediately, preparing for a write to the RAM on the next rising `clk`, and
- on the next rising edge of `clk`, `wr_addr` is incremented.

In systems where the `push_n` and/or associated `data_in` are late-arriving, the DW_lp_fifo_1c_df is configurable to add one level of re-timing registers to the both sets of signals to guarantee meeting timing specifications as configured by the *arch_type* parameter (values of 2 or 4).

For cases where *arch_type* is 2 or 4, asserting `push_n` when `full` and pop_n are not asserted causes the following to occur:

If pre-fetch cache is not full, the input register is empty, and the RAM is empty,

- `data_in` is written into the input register on the next rising edge of `clk`,
- the write address to RAM is not advanced, and
- on the subsequent rising edge of `clk`, the input register contents is written directly into the pre-fetch cache.

If pre-fetch cache has at least 2 empty locations, the input register is populated, and the RAM is empty,

- `data_in` is written into the input register on the next rising edge of `clk`,
- the write address to RAM is not advanced, and
- on the subsequent rising edge of `clk`, the input register contents is written directly into the pre-fetch cache.

If pre-fetch cache has only 1 empty location, the input register is populated, and the RAM is empty,

- `data_in` is written into the input register AND the input register contents is written into the pre-fetch cache on the next rising edge of `clk`,
- the DW_lp_fifoctl_1c_df `ram_we_n` is asserted, and
- on the subsequent rising edge of `clk`, the input register contents is written directly into the RAM with the `wr_addr` incremented.

Write Errors

An error occurs if a push operation is attempted while the FIFO is full. That is, the `error` output goes active if:

- the `push_n` input is asserted,
- the `pop_n` input is not asserted, and
- the `full` flag is active on the rising edge of `clk`.

After a push error, although a data word was lost at the time of the error, the FIFO remains in a valid full state and can continue to operate properly with respect to the data that was contained in the FIFO before the push error occurred.

Read Operations

Reading from the FIFO

Reading of the FIFO is initiated when `pop_n` is asserted while the `empty` status is not set.

If *arch_type* is 0, the `data_out` is driven directly from the RAM. Asserting `pop_n` while `empty` is not active causes the internal read pointer to increment on the next rising edge of `clk` only if the RAM contains at least one valid entry.

If *arch_type* is not 0, the `data_out` is driven from the pre-fetch cache. Depending on the state of the cache and RAM, the internal read pointer behavior to the RAM varies. If the RAM is empty during a pop request, only the pre-fetch cache contains valid data and no advancement of the internal read pointer to RAM is made. However, if during a pop request the RAM contains at least one valid data entry, the internal read pointer is incremented on the next rising edge of clk.

Popping from the Cache (Referencing Pipelined Architecture)

When the *arch_type* is not 0, the cache is the data interface of the FIFO and it is made up of data locations based on the *mem_mode* parameter as described in Table 1-7 on page 11. When the cache contains at least one valid entry the FIFO is considered not empty, that is the `empty` flag is not asserted, a legal pop of the FIFO is allowed (asserting `pop_n`). When empty is not asserted the `data_out` contents is the next valid word from the FIFO. The assertion of `pop_n` causes the cache pointer to advance to the next valid data, if any, on the next rising edge of `clk`. If valid data in RAM is available, the current RAM output data addressed by the internal read pointer is loaded into the next vacant stage of the cache.

However, if only one location of the cache contains valid data and RAM output data is not valid, `push_n` is not asserted, and `pop_n` is asserted, then on the next rising edge of `clk` that data value at that lone valid location of cache (which is driving `data_out`) is held AND the `empty` flag gets asserted. Thus, assertion of the `empty` flag declares the contents at `data_out` irrelevant.

Read Errors

An error occurs if a pop operation is attempted while the FIFO is empty. That is, the `error` output goes active if:

- the `pop_n` input is active and
- the `empty` flag is active on the rising edge of `clk`.

When a pop error occurs, the internal RAM read pointer does not advance. After a pop error, the FIFO is still in a valid empty state and can continue to operate properly.

Status Flags and Error Output

The error outputs and flags are initialized as follows:

- `empty` and `almost_empty` are initialized to 1
- All other flags and the `error` output are initialized to 0

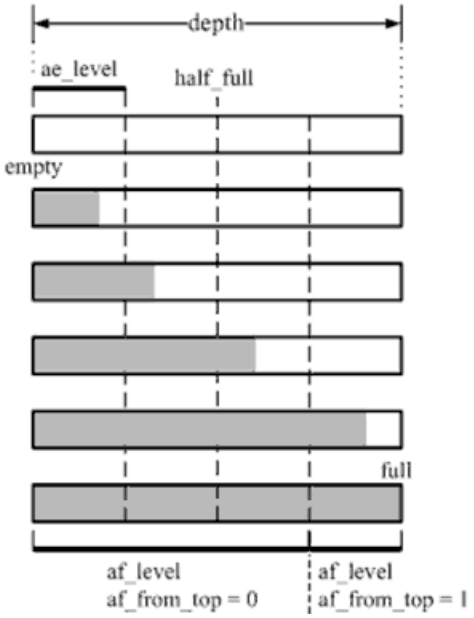
The status flags respond immediately to changes in state on the next rising edge of clock following either a push or pop operation.

The `almost_empty` flag is dependent on the input value of `ae_level`. Similarly, the `almost_full` flag is dependent on the value placed on the input `af_level` as depicted in the figure below. When `ae_level` and `af_level` are held fixed during normal operation, the `almost_empty`, `almost_full`, and the other status flags respond immediately to changes in state on the next rising edge of clock following either a push or pop operation. If, however, the FIFO is in a non-empty state and either or both of the inputs `ae_level` and `af_level` change, the corresponding state of `almost_empty` and `almost_full` will not update until either a push or pop request is issued. That is, when `ae_level` and/or `af_level` input are changed, there could be a moment in time when the `almost_empty` and/or `almost_full` flags are not accurate when compared to the word count reported.

The `almost_empty` and/or `almost_full` flags will be correct again on the next rising edge of `clk` after either `push_n` or `pop_n` (but not both) is asserted. If this behavior is not desired, the `level_change` input is provided to enable the updating of the `almost_empty` and/or `almost_full` flags on the next rising edge of `clk` following the change of the `ae_level` and/or `af_level` inputs while `level_change` is active (1).

This method of updating of status flags only when the word count changes (that is, push without pop or pop without push) is in place to minimize dynamic power in the underlying DW_lp_fifoctl_1c_df. So, providing the `level_change` input enables newly changed `ae_level` and `af_level` input values to be immediately applied to report accurate status flags without relying on active push or pop requests. However, if this behavior is not critical, tying off the `level_change` input to a logic 0 will provide better quality of results in timing critical designs. For example, if system behavior exists where `ae_level` and `af_level` values are only changed during system reset, then it is recommended that `level_change` be tied to logic 0 to provide the best possible opportunity to minimize timing through the component.

Figure 1-5 Status Flag Interpretation



Status Flags

empty	almost_empty	half_full	almost_full	full
1	1	0	0	0
0	1	0	0	0
0	0	0	0	0
0	0	1	0	0
0	0	1	1	0
0	0	1	1	1

Most status flags have a property which is potentially useful to the designed operation of the FIFO controller. These properties are described in the following explanations of the flag behaviors.

empty Status Flag

The `empty` output is active high and registered. `empty` indicates that the FIFO contains no valid data entries. During the first push the rising edge of `clk` causes the first word to be written into the pre-fetch cache (or RAM if *arch_type* is 0) and `empty` is driven low. Upon the pop of the last valid data entry (and no simultaneous push request), the `empty` is driven high on the next rising edge of `clk`.

Property of empty

If `empty` is active then the FIFO is truly empty.

almost_empty Status Flag

The `almost_empty` output is active high, registered, and indicates that the FIFO is almost empty when there are no more than `ae_level` words currently in the FIFO to be popped.

The `ae_level` input defines the almost empty threshold. The `almost_empty` output is useful when it is desirable to push data into the FIFO in bursts (without allowing the FIFO to become empty).

Property of almost_empty

If `almost_empty` is active then the FIFO has at least '`depth - ae_level`' available locations. Therefore such status indicates that the push interface can safely and unconditionally push *depth* - `ae_level` words into the FIFO. This property guarantees that such a 'blind push' operation will not overrun the FIFO.

half_full status flag

The `half_full` output is active high, registered, and indicates that the FIFO has at least half of its memory locations occupied.

Property of half_full

If `half_full` is inactive then the FIFO has at least half of its locations available. Thus such status indicates that the push interface can safely and unconditionally push 'INT(*depth*/2)' words into the FIFO. This property guarantees that such a 'blind push' operation will not overrun the FIFO.

almost_full Status Flag

The `almost_full` output active high and registered. Depending on the parameter *af_from_top* setting `almost_full` indicates either that the FIFO is almost full when there are no more than `af_level` empty locations in the FIFO (*af_from_top* = 1) or when there is at least `af_level` used locations in the FIFO (*af_from_top* = 0).

The `af_level` input port defines the almost full threshold. The `almost_full` output is useful when more than one cycle of advance warning is needed to stop the flow of data into the FIFO before it becomes full (to avoid a FIFO overrun). Also, it allows for a 'blind pop' operation since it guaranteed to have at least `af_level` entries exist for the *af_from_top* = 0 case or *depth* - `af_level` entries exist for *af_from_top* = 1.

When *af_from_top* is set to 1, a subtraction operation is performed using *depth* and `af_level`. When *af_from_top* is set to 0, no subtraction is performed using the `af_level` input.

Property of almost_full

For *af_from_top* = 0:

If `almost_full` is inactive (low) then the RAM module has at least (*depth* - `af_level` + 1) available locations. Thus such status indicates that the push interface can safely and unconditionally push (*depth* - `af_level` + 1) words into the FIFO. This property guarantees that such a 'blind push' operation will not overrun the FIFO.

For *af_from_top* = 1:

If `almost_full` is inactive (low) then the RAM module has at least (`af_level`+1) available locations. Thus such status indicates that the push interface can safely and unconditionally push (`af_level`+1) words into the FIFO. This property guarantees that such a 'blind push' operation will not overrun the FIFO.

full Status Flag

The `full` output is active high, registered, and indicates that the FIFO has valid data entries in every location. During the final push the rising edge of `clk` causes the last word to be pushed and `full` is asserted. A pop request when the FIFO is full (and no simultaneous push request) causes the `full` flag to de-assert on the next rising edge of `clk`.

Property of full

If the `full` output is active, then all available entries in the FIFO (RAM, cache, and input re-timing stage (if configured to exist)) have valid data in them.

word_cnt

The `word_cnt` output is registered and represents the number of valid data entries in the FIFO. This count includes the contents in the RAM, pre-fetching cache (if configured into the design), and the input re-timing stage (if configured into the design). Upon detection of separate push and pop events, the `word_cnt` gets updated on the next rising edge of `clk`. Simultaneous push and pop events will not change its value.

The range of `word_cnt` is from 0 to *depth*.

error Output

The `error` output can indicate that a push request was issued while the `full` output was active and no pop request (an overrun error) or that a pop request was issued while the `empty` output was active (an underrun error).

The *err_mode* parameter determines whether the `error` output remains active until reset (persistent) or for only the clock cycle(s) in which the error is detected (dynamic).

When *err_mode* = 0 at design time, persistent error flags are generated. When *err_mode* = 1 at design time, dynamic error flags are generated.

When an overrun condition occurs, the write address pointer (`wr_addr`) does not advance, and the RAM write enable (`ram_we_n`) is not activated.

Therefore, a push request that would overrun the FIFO is, in effect, rejected, and an error is generated. This guarantees that no data already in the FIFO is destroyed (overwritten). Other than the loss of the data accompanying the rejected push request, FIFO operation can continue without reset.

When an underrun condition occurs, the read address pointer (`rd_addr`) does not advance, as there is no data in the FIFO to retrieve.

The FIFO timing is such that the logic controlling the `pop_n` input would not see the error until 'nonexistent' data had already been registered by the receiving logic. This is easily avoided if this logic can pay close attention to the `empty` output and thus avoid an underrun completely.

Reset

System Resets (Synchronous and Asynchronous)

Two system resets are available: `rst_n` is asynchronous or synchronous and `init_n` is synchronous.

The *rst_mode* parameter defines whether `rst_n` behavior is asynchronous or synchronous, and also determines whether asserted *rst_n* and *init_n* clear the RAM.

If both resets are connected to active logic, `rst_n` has precedence if both resets are asserted simultaneously. If only one of the resets is active in the system, the other should be tied to the de-asserted state (logic 1).

For examples of timing waveforms where `init_n` and `rst_n` are asserted, see Figure 1-10 on page 26, Figure 1-11 on page 27 and Figure 1-12 on page 28.

Status Flags During Reset Conditions

Table 1-8 identifies the reset state of each status flag.

Table 1-8 Status Flag States during Reset Conditions

Status Flag	Value During Reset
empty	1
almost_empty	1
half_full	0
almost_full	0
full	0
word_cnt	0
error	0

Suppressing Warning Messages During Verilog Simulation

The Verilog simulation model includes macros that allow you to suppress warning messages during simulation.

To suppress all warning messages for all DWBB components, define the DW_SUPPRESS_WARN macro in either of the following ways:

- Specify the Verilog preprocessing macro in Verilog code:

  ```
  `define DW_SUPPRESS_WARN
  ```

- Or, include a command line option to the simulator, such as:

  `+define+DW_SUPPRESS_WARN` (which is used for the Synopsys VCS simulator)

The warning messages for this model include the following:

- If values other than 1 or 0 are present on a clock port, the following message is displayed:

  ```
  WARNING: <instance_path>.<clock_name>_monitor:
      at time = <timestamp>, Detected unknown value, x, on <clock_name> input.
  ```

  To suppress only this warning message for all DWBB components, use the following macro:

  - Define the DW_DISABLE_CLK_MONITOR macro. You can define this macro in the following ways:
    - Specify the Verilog preprocessing macro in Verilog code:

      ```
      `define DW_DISABLE_CLK_MONITOR
      ```

    - Or, include a command line option to the simulator, such as:

      `+define+DW_DISABLE_CLK_MONITOR` (which is used for the Synopsys VCS simulator)

  This message is also suppressed using the DW_SUPPRESS_WARN macro explained earlier.

Timing Waveforms

The following waveforms provide signal and timing behavior for typical DW_lp_fifo_1c_df operation.

Push to Full, Pop to Empty, and Pop Error

Figure 1-6 on page 20 shows the configuration where a 2-deep pre-fetch cache exists (*arch_type* is 1 and *mem_mode* is 2).

Configuration: *width* is 4, *depth* is 6, *mem_mode* is 2, *arch_type* is 1, *af_from_top* is 0, *ram_re_ext* is 1, and *err_mode* is 0; 2-deep PL pre-fetch cache, read address re-timing stage in RAM.

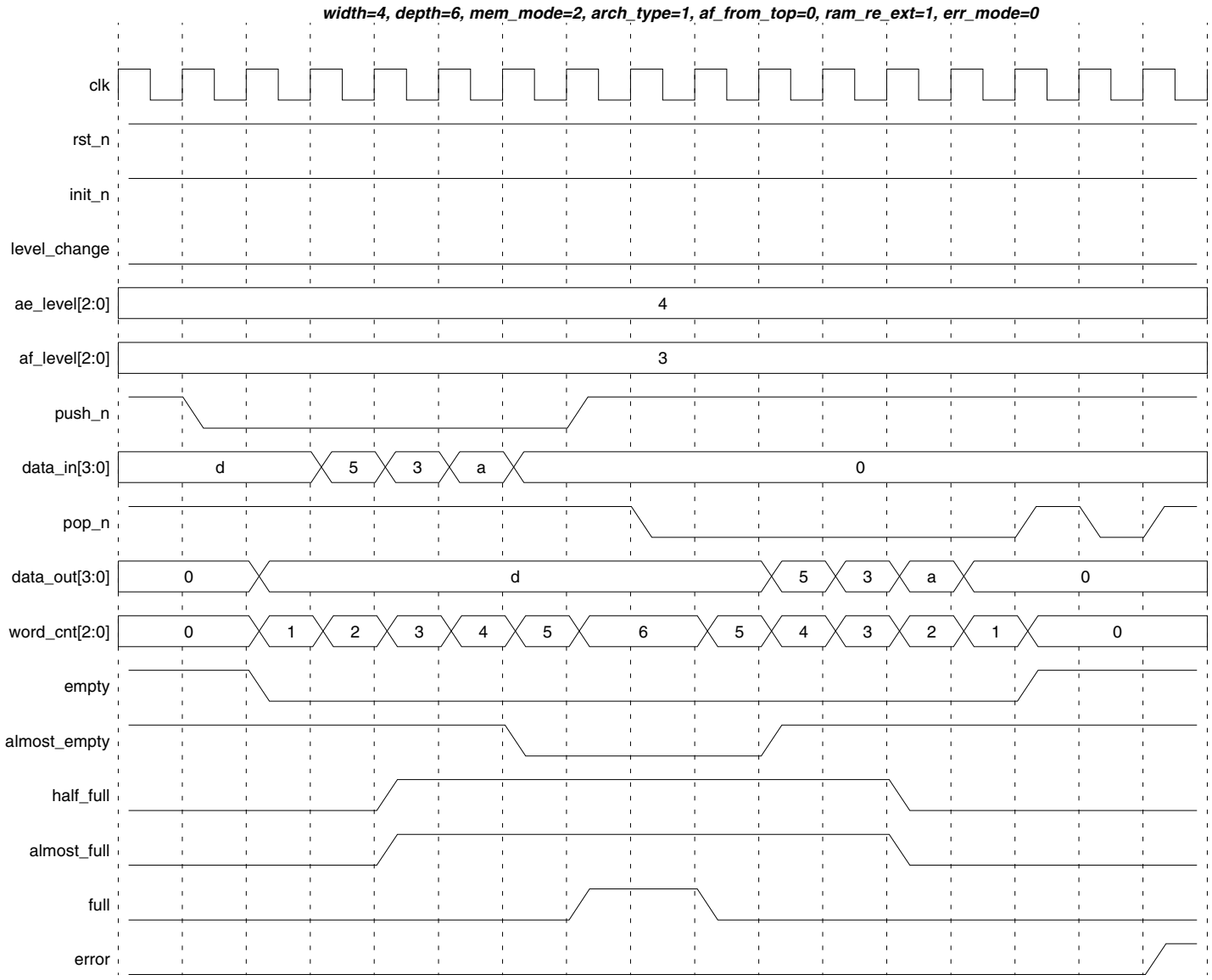
In this timing waveform, the FIFO is pushed to a full state followed by popping to empty. In the empty state, a pop request is made with caused the `error` output to assert. Note that at the pop error condition, `data_out` holds its previous value.

With regard to the other status flags, the `almost_empty` and `almost_full` outputs are configured based on the inputs `ae_level` and `af_level`, respectively. Additionally for the `almost_full` flag, the parameter *af_from_top* determines how the `af_level` input value is interpreted.

In this example below, *af_from_top* is 0 which means that whatever value is driven on `af_level` will be the threshold at which the `almost_full` flag goes to 1. Any word count less than `af_level` will result in `almost_full` being a 0. In these waveforms `af_level` is 3. So, whenever `word_cnt` is 3 or greater, `almost_full` is 1.

The `almost_empty` flag always uses the same interpretation of its corresponding `ae_level` input signal value. Whenever the word count in the FIFO is less than or equal to `ae_level`, `almost_empty` is 1. In this waveform below, `ae_level` is 4. So, as long as the `word_cnt` is 4 or less, `almost_empty` is 1.

Figure 1-6 Push to Full, Pop to Empty, and Pop Error



Push to Full, Push and Pop While Full, Push Error

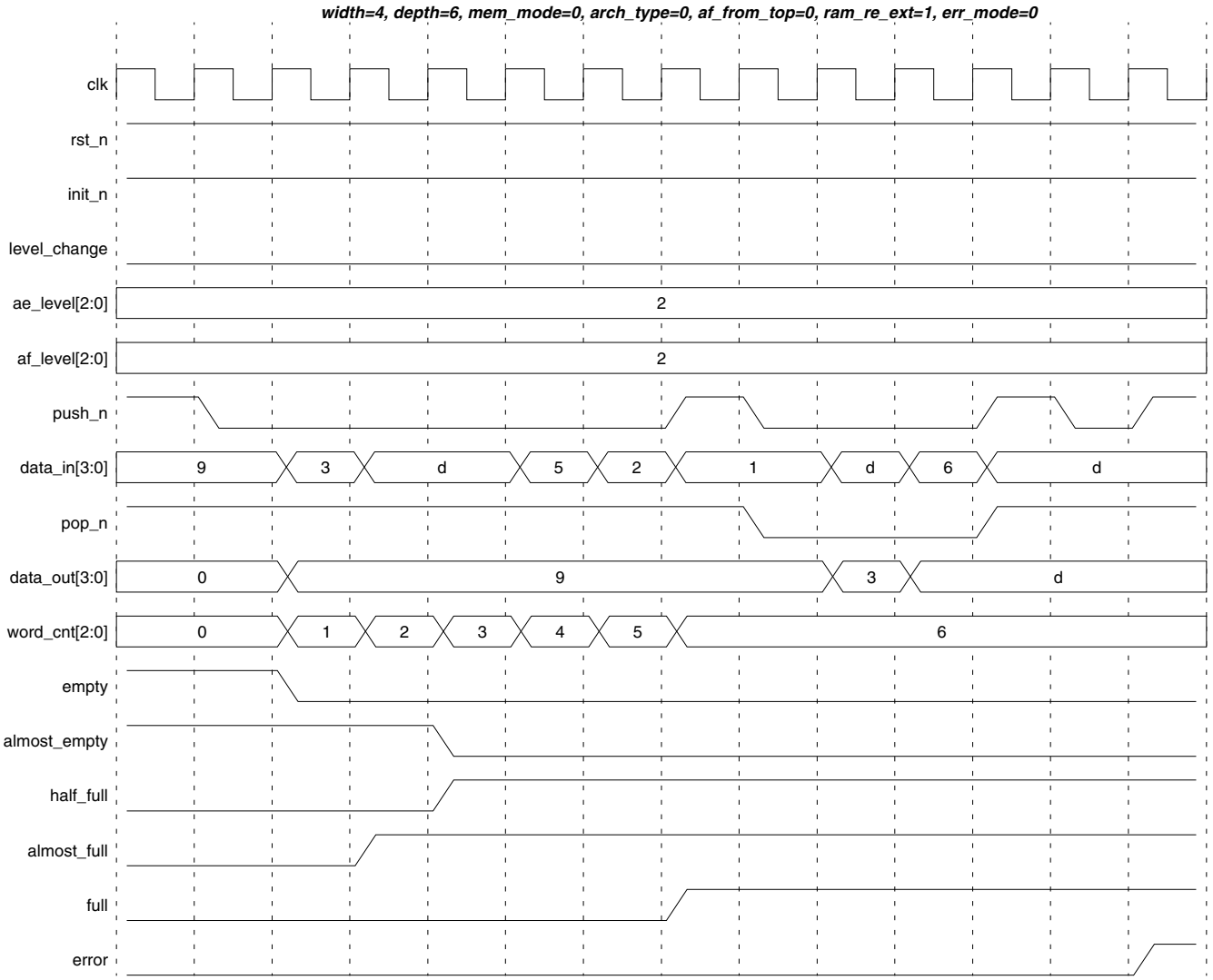
Figure 1-7 illustrates continuously pushing of the FIFO until full from the empty state, multiple cycles of simultaneous push and pop requests during `full` condition, and a push error.

Configuration: *width* is 4, *depth* is 6, *mem_mode* is 0, *arch_type* is 0, *af_from_top* is 0, *ram_re_ext* is 1, and *err_mode* is 0; no pre-fetch cache and no re-timing stages in RAM

In this example below, *af_from_top* is 0. This means that whatever value is driven on `af_level` will be the threshold at which the `almost_full` flag goes to 1. Any word count less than `af_level` will result in `almost_full` being a 0. In these waveforms `af_level` is 2. So, whenever `word_cnt` is 2 or greater, `almost_full` is 1.

The `almost_empty` flag always uses the same interpretation of its corresponding `ae_level` input signal value. Whenever the word count in the FIFO is less than or equal to `ae_level`, `almost_empty` is 1. In this example below, `ae_level` is 2. So, as long as the `word_cnt` is 2 or less, `almost_empty` is 1.

Figure 1-7 Push to Full, Push and Pop While Full, Push Error



Push and Pop on Empty FIFO

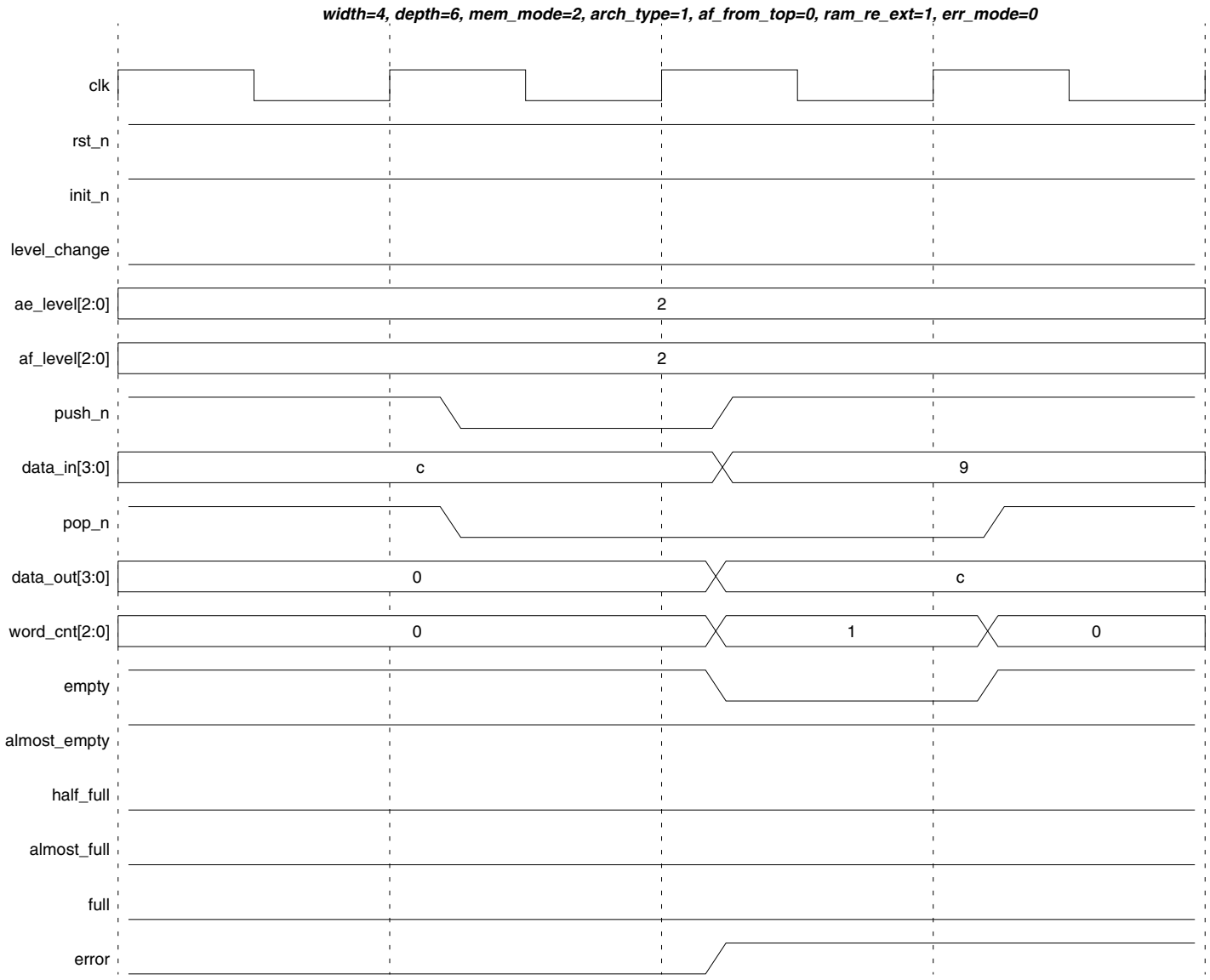
Figure 1-8 on page 23 illustrates the condition in which the FIFO is empty and push and pop requests occur simultaneously.

Configuration: *width* is 4, *depth* is 6, *mem_mode* is 2, *arch_type* is 1, *af_from_top* is 0, *ram_re_ext* is 1, and *err_mode* is 0; PL pre-fetch cache depth of 2, read address re-timing stage in RAM.

For this particular sequence, the pop request on an empty FIFO causes the `error` flag to go active. However, the push request is legal which results in the `data_in` value of '0xc' being written into the first (and only) stage of the pre-fetch cache. Therefore, on the next clock cycle following the sampled assertion of `pop_n`, the `data_out` contains the '0xc' value. As a result, the `word_cnt` value is updated to 1. Note that the `error` flag once set to 1 stays asserted even though the 'illegal' pop request occurs for only one clock cycle. This is due to the *err_mode* setting of 0 which configures the `error` flag to maintain its asserted state until a system reset is performed.

Since the push request was allowed and `data_in` was written into cache, the second `pop_n` assertion (immediately following the illegal pop request) is legal and pops the word from the FIFO (or more specifically, from cache). This results in `word_cnt` going back to 0 and the FIFO `empty` status flag going active.

Figure 1-8 Push and Pop on Empty FIFO (‘data_in’ Is Still Captured)



Effects of ae_level, af_level, and level_change

Figure 1-9 on page 24 describes the affects changing 'ae_level', 'af_level', and 'level_change' have on the 'almost_empty' and 'almost_full' flags.

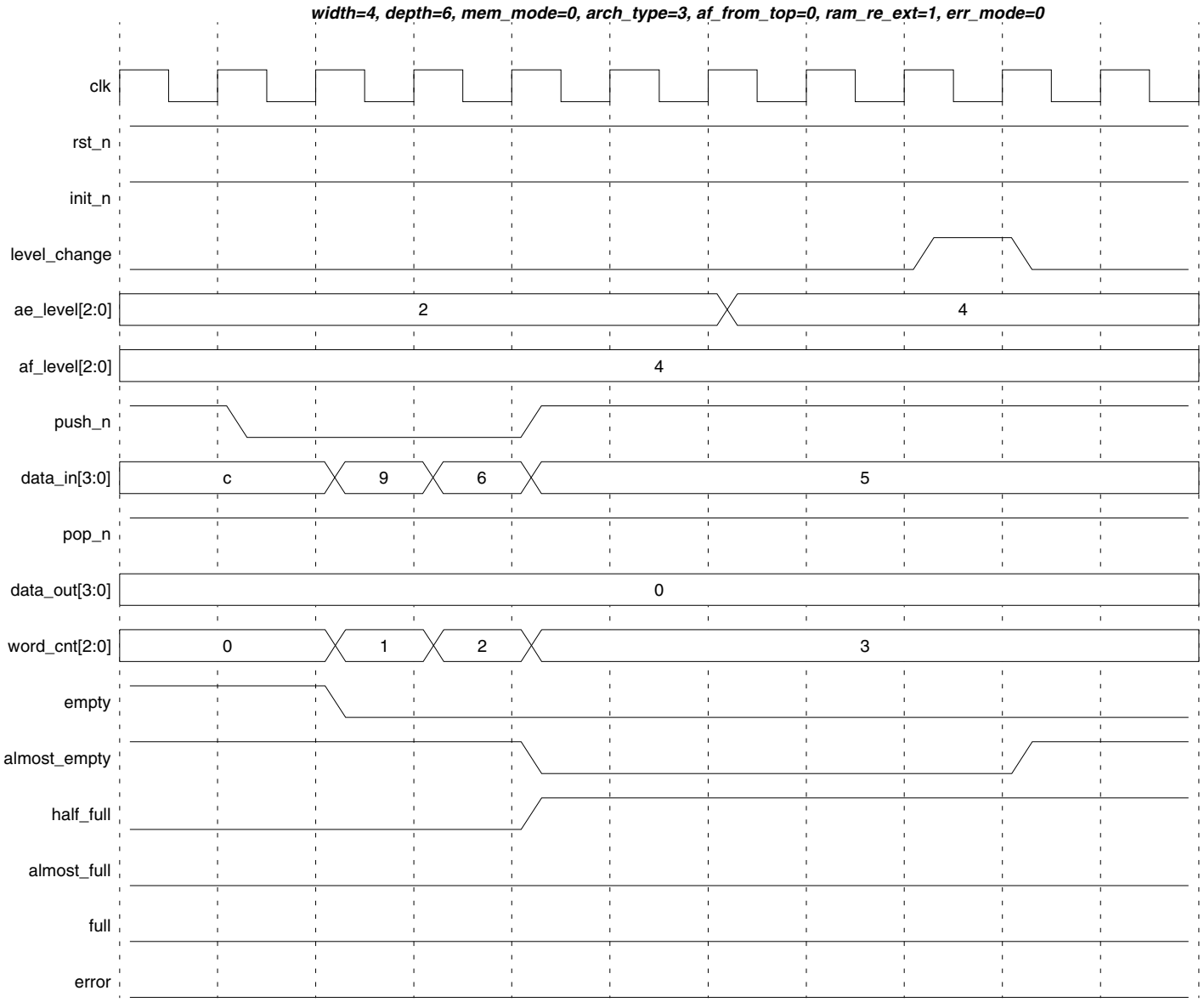
Configuration: *width* is 4, *depth* is 6, *mem_mode* is 0, *arch_type* is 3, *af_from_top* is 0, *ram_re_ext* is 1, and *err_mode* is 0; RF pre-fetch cache depth of 3, read address re-timing and `data_out` stages in RAM.

Initially, the `ae_level` setting is 2 and `af_level` is 4. From the FIFO empty state, three active cycles of `push_n` result in the FIFO to have three words written into it. This is reflected by `word_cnt` settling on 3. Since `ae_level` is 2, the `almost_empty` flag stays asserted while `word_cnt` is 2 and less. Upon the `word_cnt` going to 3, `almost_empty` de-asserts.

After the three consecutive push requests, the FIFO remains idle until `ae_level` changes from 2 to 4. The change in `ae_level` alone does not render a change in the `almost_empty` flag as evident in the `word_cnt` compared to the `ae_level` value not matching the state of the `almost_empty` flag in the cycles that follow. If the state of the `almost_empty` flag is critical between push and pop operations on a non-empty FIFO, the `level_change` input causes the `almost_empty` (and `almost_full`) flag to get updated on the next clock cycle. This is clearly seen in the waveform below once the 'level_change' input is sampled to be 'high'. The `almost_empty` flag goes to '1' on the following clock cycle to reflect the state brought on by the newly changed value of 4 on the `ae_level` input.

This example shows how the use of `level_change` updates the `almost_empty` and `almost_full` flags should the `ae_level` and `af_level` inputs change. If the `ae_level` and `af_level` inputs change but `level_change` is not asserted, then the `almost_empty` and `almost_full` flags are updated according the 'new' level values on the next push and/or pop request.

Figure 1-9 Effects of ae_level, af_level, and level_change



'init_n' When '*rst_mode*' is '2' (Clears RAM)

Figure 1-10 on page 26 shows the affects of 'init_n' on RAM and sequential elements for the configuration where no pre-fetch cache exists (*arch_type* = '0'). When '*arch_type*' is '0', '*mem_mode*' must also be '0'.

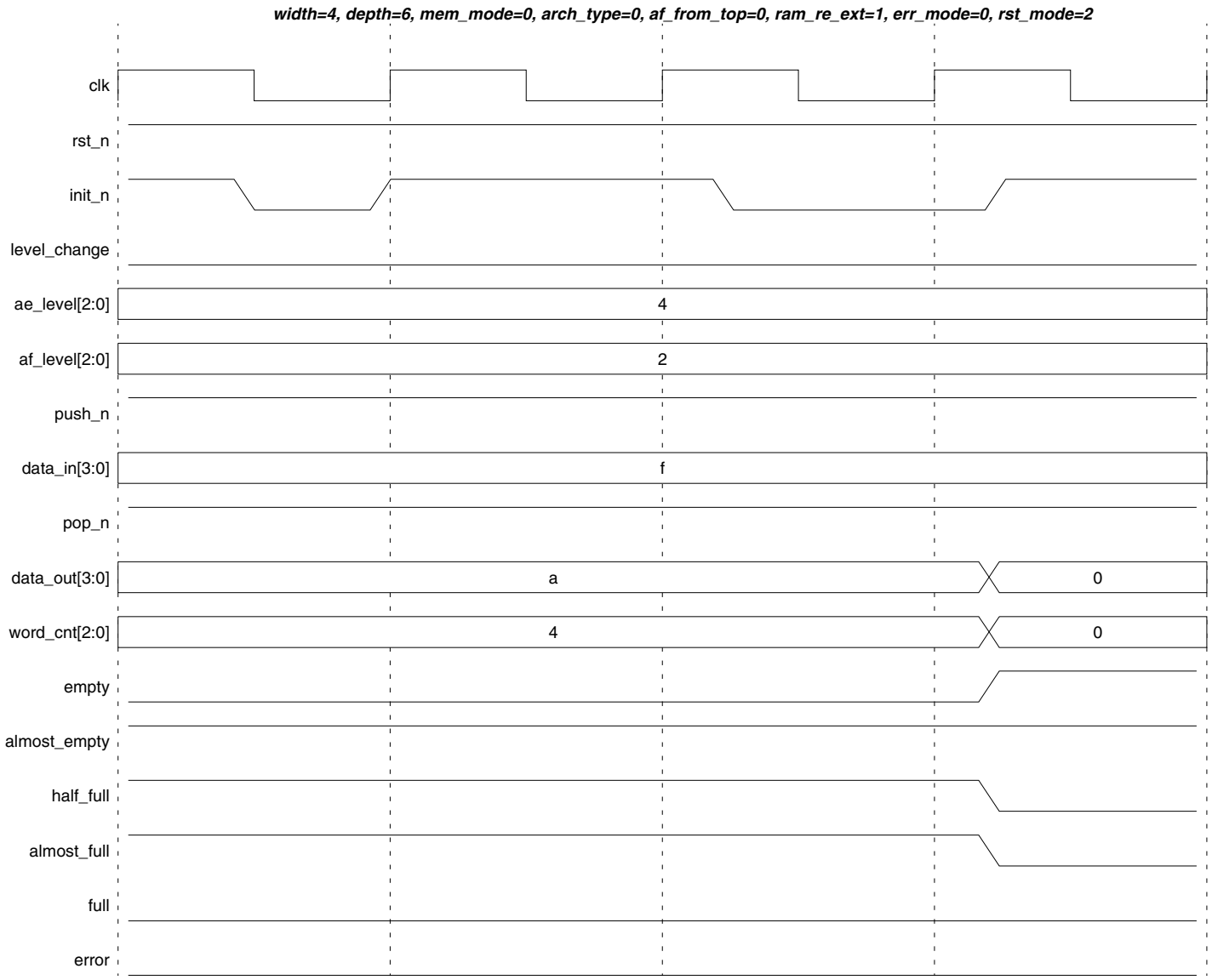
Configuration: *width* is 4, *depth* is 6, *mem_mode* is 0, *arch_type* is 0, *af_from_top* is 0, *ram_re_ext* is 1, *err_mode* is 1, and *rst_mode* is 2; no pre-fetch cache, no re-timing stages in RAM.

With *rst_mode* set to 2, the init_n clears RAM when asserted.

To illustrate that RAM does get cleared when *rst_mode* is 2, a configuration with *arch_type* set to 0 is used since this does not employ a pre-fetch cache which means data_out of DW_lp_fifo_1c_df is a direct connection from the data output of the RAM.

The init_n is the synchronous reset for DW_lp_fifo_1c_df. Only when init_n is 0 and captured by the rising edge of clk will the DW_lp_fifo_1c_df sequential elements get initialized and, in this case, RAM is cleared as well; this is shown by two assertions of init_n in the following figure. The first occurrence of init_n going to 0 does not get sampled by the rising edge of clk. Therefore, the sequential elements in the DW_lp_fifo_1c_df are unaffected and not state change occurs. However, the second assertion of init_n spans across the rising edge boundary of clk which causes all the registers and RAM elements to be initialized to their default values. Most notably, word_cnt goes to 0, half_full and almost_full go to 0, and empty and almost_empty go to 1, and data_out goes to 0.

Figure 1-10 init_n When *rst_mode* is 2 (Clears RAM)



rst_n When *rst_mode* Is 1 (RAM Not Cleared)

Figure 1-11 on page 27 shows the affects of `rst_n` on RAM for the configuration where no pre-fetch cache exists (*arch_type* is 0). When *arch_type* is 0, *mem_mode* must also be 0.

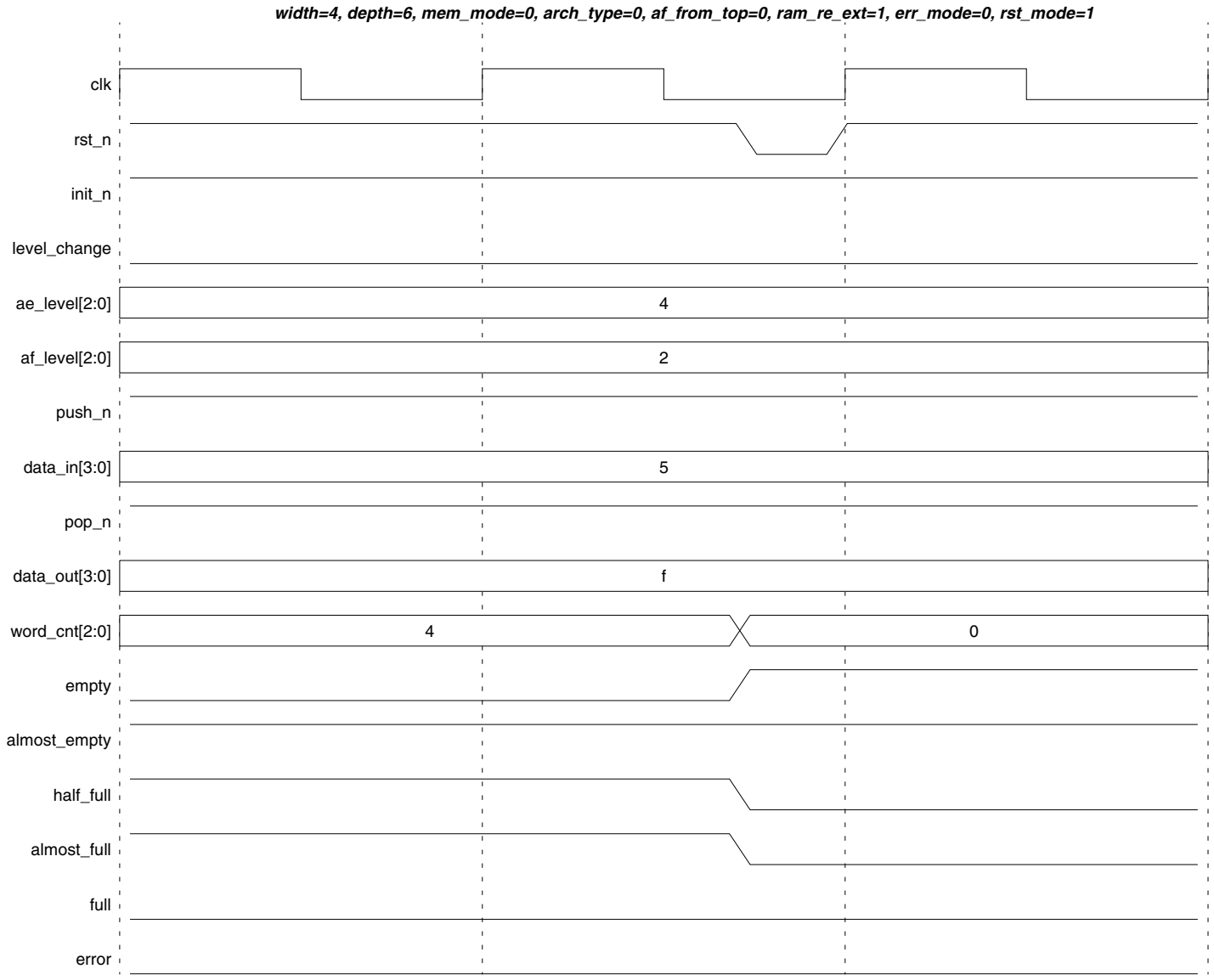
Configuration: *width* is 4, *depth* is 6, *mem_mode* is 0, *arch_type* is 0, *af_from_top* is 0, *ram_re_ext* is 1, *err_mode* is 1, and *rst_mode* is 1; no pre-fetch cache, no re-timing stages in RAM.

With *rst_mode* set to 1, the `rst_n` does not clear RAM when asserted.

To illustrate that RAM does not get cleared when *rst_mode* is 1, a configuration with *arch_type* set to 0 is used since this does not employ a pre-fetch cache which means `data_out` of DW_lp_fifo_1c_df is a direct connection from the data output of the RAM. So, any assertion of `rst_n` applied will not cause `data_out` to change state.

The `rst_n` is an asynchronous reset for DW_lp_fifo_1c_df in this case with *rst_mode* being 1. Whenever `rst_n` goes to 0, the DW_lp_fifo_1c_df sequential elements get initialized except for RAM elements. From the waveforms below, as soon as the `rst_n` goes to 0 all the registers to be initialized to their default values. Most notably, `word_cnt` goes to 0, `half_full` and `almost_full` go to 0, and `empty` and `almost_empty` go to 1. Also note that `data_out` holds its previous value.

Figure 1-11 rst_n When *rst_mode* is 1 (RAM Not Cleared)



rst_n When *rst_mode* Is 2 (Clears RAM)

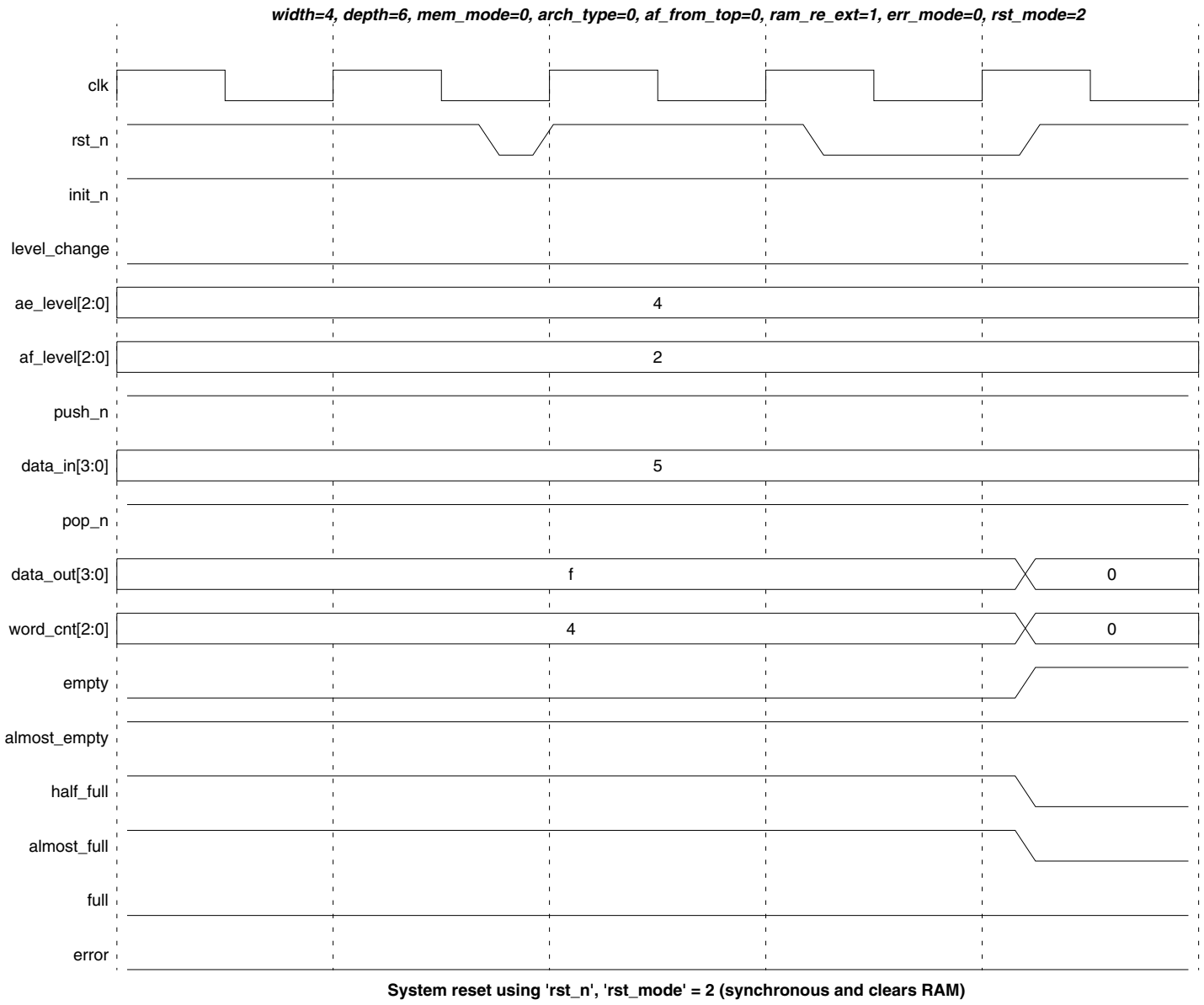
Figure 1-12 on page 28 shows the affects of `rst_n` on RAM and sequential elements for the configuration where no pre-fetch cache exists (*arch_type* is 0). When *arch_type* is 0, *mem_mode* must also be 0.

Configuration: *width* is 4, *depth* is 6, *mem_mode* is 0, *arch_type* is 0, *af_from_top* is 0, *ram_re_ext* is 1, *err_mode* is 1, and *rst_mode* is 2; no pre-fetch cache, no re-timing stages in RAM.

With *rst_mode* set to 2, the `rst_n` is synchronous and clears RAM when asserted. To show this, a configuration with *arch_type* set to 0 is used since this does not employ a pre-fetch cache which means `data_out` of DW_lp_fifo_1c_df is a direct connection from the data output of the RAM.

The `rst_n` input is a synchronous reset in this case because *rst_mode* is 2. Only when `rst_n` is 0 and captured by the rising edge of `clk` will the DW_lp_fifo_1c_df sequential elements get initialized and, in this case, RAM is cleared as well; this is shown by two assertions of `rst_n` in the following figure. The first occurrence of `rst_n` going to 0 does not get sampled by the rising edge of `clk`. Therefore, the sequential elements in the DW_lp_fifo_1c_df are unaffected and no state change occurs. However, the second assertion of `rst_n` spans across the rising edge boundary of `clk`, which initializes all registers and RAM elements to their default values. Most notably, `word_cnt` goes to 0, `half_full` and `almost_full` go to 0, and `empty` and `almost_empty` go to 1, and `data_out` goes to 0.

Figure 1-12 rst_n When *rst_mode* Is 2 (Clears RAM)



Related Topics

- Memory – FIFO Overview
- DesignWare Building Block IP User Guide

HDL Usage Through Component Instantiation - VHDL

```
library IEEE,DWARE;
use IEEE.std_logic_1164.all;
use DWARE.DWpackages.all;
use DWARE.DW_Foundation_comp.all;

entity DW_lp_fifo_1c_df_inst is
      generic (
        inst_width : POSITIVE := 8;
        inst_depth : POSITIVE := 8;
        inst_mem_mode : NATURAL := 3;
        inst_arch_type : NATURAL := 1;
        inst_af_from_top : NATURAL := 1;
        inst_ram_re_ext : NATURAL := 0;
        inst_err_mode : NATURAL := 0;
        inst_rst_mode : NATURAL := 0
        );
      port (
        inst_clk : in std_logic;
        inst_rst_n : in std_logic;
        inst_init_n : in std_logic;
        inst_ae_level : in std_logic_vector(3 downto 0);
        inst_af_level : in std_logic_vector(3 downto 0);
        inst_level_change : in std_logic;
        inst_push_n : in std_logic;
        inst_data_in : in std_logic_vector(inst_width-1 downto 0);
        inst_pop_n : in std_logic;
        data_out_inst : out std_logic_vector(inst_width-1 downto 0);
        word_cnt_inst : out std_logic_vector(3 downto 0);
        empty_inst : out std_logic;
        almost_empty_inst : out std_logic;
        half_full_inst : out std_logic;
        almost_full_inst : out std_logic;
        full_inst : out std_logic;
        error_inst : out std_logic
        );
    end DW_lp_fifo_1c_df_inst;

architecture inst of DW_lp_fifo_1c_df_inst is

begin
    -- Instance of DW_lp_fifo_1c_df
```

```
    U1 : DW_lp_fifo_1c_df
    generic map ( width => inst_width,
                      depth => inst_depth,
                      mem_mode => inst_mem_mode,
                      arch_type => inst_arch_type,
                      af_from_top => inst_af_from_top,
                      ram_re_ext => inst_ram_re_ext,
                      err_mode => inst_err_mode,
                      rst_mode => inst_rst_mode
                    )
    port map ( clk => inst_clk,
                   rst_n => inst_rst_n,
                   init_n => inst_init_n,
                   ae_level => inst_ae_level,
                   af_level => inst_af_level,
                   level_change => inst_level_change,
                   push_n => inst_push_n,
                   data_in => inst_data_in,
                   pop_n => inst_pop_n,
                   data_out => data_out_inst,
                   word_cnt => word_cnt_inst,
                   empty => empty_inst,
                   almost_empty => almost_empty_inst,
                   half_full => half_full_inst,
                   almost_full => almost_full_inst,
                   full => full_inst,
                   error => error_inst
                 );


end inst;

-- Configuration for use with a VHDL simulator
-- pragma translate_off
library DW03;
configuration DW_lp_fifo_1c_df_inst_cfg_inst of DW_lp_fifo_1c_df_inst is
  for inst
  end for; -- inst
end DW_lp_fifo_1c_df_inst_cfg_inst;
-- pragma translate_on
```

HDL Usage Through Component Instantiation - Verilog

```
module DW_lp_fifo_1c_df_inst ( inst_clk, inst_rst_n, inst_init_n,
           inst_ae_level, inst_af_level, inst_level_change, inst_push_n,
           inst_data_in, inst_pop_n, data_out_inst, word_cnt_inst,
           empty_inst, almost_empty_inst, half_full_inst, almost_full_inst,
           full_inst, error_inst
                          );

parameter width       = 8;
parameter depth       = 8;
parameter mem_mode    = 3;
parameter arch_type   = 1;
parameter af_from_top = 1;
parameter ram_re_ext  = 0;
parameter err_mode    = 0;
parameter rst_mode    = 0;

`define cnt_width   4  // log2(depth+1)

input                           inst_clk;
input                           inst_rst_n;
input                           inst_init_n;
input  [`cnt_width-1:0]         inst_ae_level;
input  [`cnt_width-1:0]         inst_af_level;
input                           inst_level_change;
input                           inst_push_n;
input  [width-1:0]              inst_data_in;
input                           inst_pop_n;

output [width-1:0]              data_out_inst;
output [`cnt_width-1:0]         word_cnt_inst;
output                          empty_inst;
output                          almost_empty_inst;
output                          half_full_inst;
output                          almost_full_inst;
output                          full_inst;
output                          error_inst;



DW_lp_fifo_1c_df #(width, depth, mem_mode, arch_type, af_from_top, ram_re_ext,
err_mode, rst_mode) U1 (
            .clk(inst_clk),
            .rst_n(inst_rst_n),
            .init_n(inst_init_n),
            .ae_level(inst_ae_level),
            .af_level(inst_af_level),
            .level_change(inst_level_change),
```

```
                .push_n(inst_push_n),
                .data_in(inst_data_in),
                .pop_n(inst_pop_n),
                .data_out(data_out_inst),
                .word_cnt(word_cnt_inst),
                .empty(empty_inst),
                .almost_empty(almost_empty_inst),
                .half_full(half_full_inst),
                .almost_full(almost_full_inst),
                .full(full_inst),
                .error(error_inst)
                );

  endmodule
```

Revision History

For notes about this release, see the *DesignWare Building Block IP Release Notes*.

For lists of both known and fixed issues for this component, refer to the STAR report.

For a version of this datasheet with visible change bars, click here.

Date	Release	Updates
July 2020	DWBB_201912.5	■ Adjusted content and title of "Suppressing Warning Messages During Verilog Simulation" on page 18 and added the DW_SUPPRESS_WARN macro
October 2019	DWBB_201903.5	■ Added the "Disabling Clock Monitor Messages" section
March 2019	DWBB_201903.0	■ Clarified some information about minPower in Table 1-3 on page 4 ■ Added this Revision History table and the document links on this page

Copyright Notice and Proprietary Information

Destination Control Statement

All technical data contained in this publication is subject to the export control laws of the United States of America. Disclosure to nationals of other countries contrary to United States law is prohibited. It is the reader's responsibility to determine the applicable regulations and to comply with them.

Disclaimer

SYNOPSYS, INC., AND ITS LICENSORS MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Trademarks

Synopsys and certain Synopsys product names are trademarks of Synopsys, as set forth at https://www.synopsys.com/company/legal/trademarks-brands.html.

All other product or company names may be trademarks of their respective owners.

Free and Open-Source Software Licensing Notices

If applicable, Free and Open-Source Software (FOSS) licensing notices are available in the product installation.

Third-Party Links

Any links to third-party websites included in this document are for your convenience only. Synopsys does not endorse and is not responsible for such websites and their practices, including privacy practices, availability, and content.

Synopsys, Inc.
www.synopsys.com